RePEAtO: Relational Paragraph-level Embeddings for Article Outlining

Michael Tang[1] **Evan Dogariu[1]** **Jiatong Yu[1]**
[1]Department of Computer Science, Princeton University
{mwtang, edogariu, jiatongy}@princeton.edu

Abstract

We formulate a novel task: article outlining, the reconstruction of the hierarchical structure of headings and subheadings within an article from just a list of paragraphs. We propose and analyze the adJacent Paragraph Least-common-ancestor-distance (JPL) score evaluation metric, compose a full-stack web application to collect human performance data on the article outlining task, and then explore various models under a composable encoder-decoder general architecture. We find that out of various embedding techniques based on past work on word, sentence, and document embeddings, SimCSE [2] performs the best and results in our strongest model using a recursive MLP decoder and paragraph embeddings composed of averaged SimCSE sentence embeddings. We find evidence that recursive models best capture the hierarchical information needed for this task, and perform ablations and further analysis of the relevant paragraph embedding spaces. Nevertheless, we find that humans far outperform our best models, and thus lay the groundwork for further study in this task, and by extension, in hierarchical tasks that require embeddings of longer forms of text.

1 Introduction

Past work in NLP representation learning has focused heavily on word-level, and more recently, sentence-level embeddings. Paragraphs encode the flow of information across individual sentences when illustrating more complex ideas – thus, we are interested in exploring ways of encoding at the paragraph level.

We propose the task of *article outlining*, predicting the nested outline of an article given only the raw paragraph sequence. We will focus on learning the structure rather than the names of the headings.

Possible applications include generating outlines of freeform notes or reconstructing the written structure of articles/transcriptions when communicated verbally. We hope paragraph-level understanding will also help future models interact with text in more human-like ways, such as retriever systems that can efficiently search through articles using just the subheadings.

2 Related Work

An important related topic is the framework of encoder-decoder models. Encoders embed paragraphs into vectors in a meaningful embedding space, which should provide useful information regarding paragraph relationships for the decoder. The embedding approaches we use are based on several different popular natural language embedding techniques, which we discuss below.

2.1 FastText

FastText is an open source library built upon Bojanowski et al.'s word embedding model[1]. The main advantage of FastText is its tolerance of out-of-vocabulary tokens. FastText takes into account morphological structures by summing over n-grams of different size. Thus it can infer unseen words from known subwords. This behavior is particularly desirable, since the out-of-vocabulary tokens can be named entities, technical words, and other things that are useful for understanding paragraph relations.

In order to generate a paragraph embedding in a fixed-dimensional latent space from a variable length sequence of words, we apply a LSTM projection layer. We discuss the implementation of this further in 3.4.

2.2 SimCSE

SimCSE is a contrastive learning-based model for state-of-the-art sentence-level embedding[2]. SimCSE applies a contrastive objective that pulls together positive-pair sentence vectors and pushes apart others to fine-tune a pretrained BERT model :

$$l_i = -\,log\,\frac{e^{sim(h_i,h_i^+)}}{\sum_j e^{sim(h_i,h_j^+)}}$$

In unsupervised training, positive pairs are generated from identical sentences inferenced with different dropout masks, while negative pairs are other sentences in each batch. The supervised SimCSE was trained on an NLI dataset, from which implications generate positive pairs and contradictions make hard negative pairs. We build off of the pre-trained supervised SimCSE model that fine-tuned BERT. Similarly to the FastText embeddings, we project sentence-level embeddings to paragraph-level embeddings (see 3.4).

2.3 Doc2Vec

Doc2Vec[4] embeds variable-length paragraphs into fixed-length vectors. Doc2Vec generates both paragraph and word vectors during training, but uses only the paragraph vector in order to embed paragraphs. During training, Doc2Vec concatenate paragraph vectors with sampled word vectors within the paragraph, and use the resulting matrix to predict next words. Here word vectors are fixed and stored in a vocabulary.

Doc2Vec is directly trained on paragraphs and produces fixed-length paragraph embeddings, which allow us to experiment with both the performance of embeddings at different scales and to reason about the performance of projection layers such as the LSTM mentioned above.

3 Methods

3.1 Task formalization

We formulate the problem as the prediction of the *heading tree* — the nested n-ary tree of headings — from a list of raw paragraphs in an article. For training, we can formulate the output as an *outline vector* or *segmentation vector*. The former encodes the indices of all of the nested headings that each paragraph falls under, while the latter succinctly encodes whether a paragraph begins or ends a section at each depth (see Figure 1). Note that both of these representations provide sufficient information to reconstruct the heading tree in full.

3.2 Task evaluation

We attempt to devise a metric for this novel task. The goal is to evaluate different predicted heading trees against a given ground truth heading tree. We specifically want the following properties:

- Fairly compares proposed heading trees with different numbers of headings and different tree depths (i.e. does not discard or underweight any nodes in a tree with more or deeper headings)
- Normalizes between articles with different numbers of paragraphs
- Independent from the heading tree representation (i.e. only engages with implicit properties of the tree structure, so it does not unfairly favor training objectives that use certain representations)

We compare four different metrics, described below.

All-Pairs Paragraph LCA (APPL) score

$$\text{APPL}(y,\hat{y},n) = \frac{1}{\binom{n}{2}} \sum_{i\neq j\in[n]} (\text{LCA}_{ij}^{\hat{y}} - \text{LCA}_{ij}^{y})^2$$

where n is the number of paragraphs in the article, $y, \hat{y}$ are the ground truth and predicted heading trees, respectively, and

$$\text{LCA}_{ij}^{t} = \text{argmin}_k[d^t(i,k)) + d^t(j,k)]$$

is the least common ancestor distance between paragraphs i, j in tree t with distance function d^t. This metric more highly weights the proper prediction of relationships between paragraphs that are further away from each other, e.g. in long sections with many paragraphs at the same depth, since it is a global metric over the entire article.

Adjacent Paragraph LCA (JPL) score

$$\text{JPL}(y,\hat{y},n) = \frac{1}{n-1} \sum_{i\neq j\in[n]} (\text{LCA}_{ij}^{\hat{y}} - \text{LCA}_{ij}^{y})^2$$

with $n, y, \hat{y}$, LCA defined the same way as in the previous section. This metric is motivated by a sample human strategy, which is to predict heading relationships by comparing neighboring paragraphs.

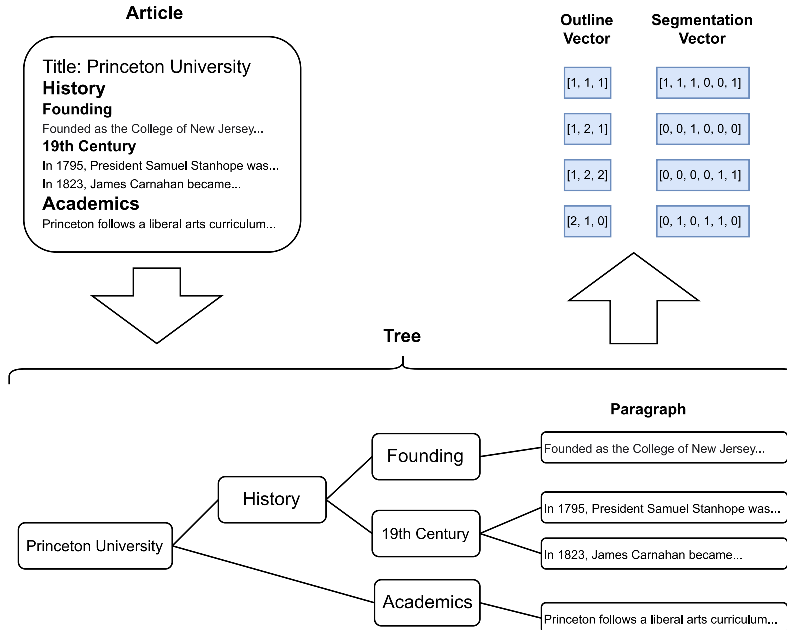


Figure 1: Article tree and respective outline vector and segmentation vector representations. For example, the third outline vector $[1, 2, 2]$ specifies that the third paragraph is the second depth-3 subheading under the second depth-2 subheading under the first depth-1 heading, and the third segmentation vector $[0, 0, 0, 0, 1, 1]$ specifies that the third paragraph does not start a section at any depth but ends a section at depths 2 and 3.

Elastic Paragraph LCA (EPL) score

$$\text{EPL}_\alpha(y, \hat{y}, n) = \alpha\text{APPL}(y, \hat{y}, n) + (1 - \alpha)\text{JPL}(y, \hat{y}, n)$$

with $n, y, \hat{y}$, LCA defined the same way as in the previous sections. This metric is motivated by Elastic Net loss, here interpolating between APPL and JPL metrics.

Edit score This is the edit distance between the bracket-symbol representations of two heading trees, where we compose string using some dummy symbol "x" to denote each paragraph and brackets to denote the sections. (e.g. [x[[xx]x][xxx]]) This metric is motivated by the way we concisely represent the heading trees when printing verbose output or in the human evaluation game (see 3.4).

Note that these metrics may be generalized to other n-ary tree structural learning problems.

3.3 Dataset

Our text data source is the Wikitext [3] dataset maintained by Huggingface, containing approximately 30,000 Wikipedia articles. We preprocess this into an article-outlining dataset, where each example consists of a flattened list of paragraphs and each label consists of the heading tree with paragraphs as the leaves. We exclude headings with no paragraph content, such as ones containing only tables.

3.4 Model architectures

We approach this problem with an *encoder-decoder* framework, where we use the embedding techniques discussed in 2 (e.g. Doc2Vec, SimCSE, FastText) to encode paragraphs into vectors in a useful latent space. We propose 3 different decoder architectures and techniques, all of which predict a heading tree from a sequence of embedded paragraphs.

Greedy decoder. Given fixed-length paragraph embeddings, we compute pairwise similarities and recursively group together pairs with similarity above a tuned threshold. We then build the tree with a bottom up dynamic programming approach based on these similarity groupings.

The full algorithm is as follows: The parameters are the input paragraphs $(X_1, ..., X_n)$, the max possible depth over articles in the dataset D, the hyperparameter thresholds for each depth $\alpha_1, ..., \alpha_D$, functions $Embed()$ and $Similarity()$, and a tree represen-

Algorithm 1 Greedy decoding

Require: $(X_1, ..., X_n), D, (\alpha_1, ..., \alpha_D)$, Embed(), Similarity(), Combine(), Node(text='', children=())
  $T \leftarrow$ (Node$(text = X_1), ...,$ Node$(text = X_n))$
  $E \leftarrow$ (Embed$(X_1), ...,$ Node$(X_n))$
  for $d \in D, ..., 0$ **do**
    $I \leftarrow ((1))$
    for $i \in 2, ..., |E|$ **do**
      if Similarity$(E_{i-1}, E_i) \geq \alpha_i$ **then**
        Add i to last section in I
      else
        Add a new section (i) to the end of I
      end if
    end for
    $T \leftarrow$ (Node(chd $= T_{I_1}), ...,$ Node(chd $= T_{I_{|I|}}))$
    $E \leftarrow$ (Combine$(I_1), ...,$ Combine$(I_{|I|}))$
    $D \leftarrow D - 1$
  end for
  return Node(chd=T)

tation $Node()$. During the algorithm, I represents the current groups of indices representing the sections at depth d of the proposed heading tree, and E gives the embedding for each section.

For the embedding function $Embed()$, we applied 4 different variations of commonly used embedding frameworks:

- LSTM-projected FastText word embeddings
- Averaged SimCSE sentence embeddings
- LSTM-projected SimCSE sentence embeddings
- 256-dimensional Doc2Vec paragraph embeddings

The two projection LSTMs are trained jointly with the Recursive MLP model detailed below. In particular, we use the training objective of the MLP to train both the MLP and 1-layer projection LSTMs for FastText word embeddings and SimCSE sentence embeddings.

For the embedding combination function $Combine()$, we simply take the element-wise vector mean of all the embeddings in the section to serve as the section embedding, and for the similarity function $Similarity()$ we use cosine similarity.

During evaluation, we used a conservative max depth value of $D = 8$, which was higher than all articles in the dataset, and individually tuned the threshold hyperparameters α_i using random search.

Recursive MLP. Given fixed-length paragraph embeddings, we use a context window to encode information about a target paragraph. We pass this into a multilayer perceptrion (MLP) feed-forward network to predict whether the center of the window marks the beginning of a section at a given depth (the depth is passed as input to each layer of the MLP). In particular, given paragraph embeddings of dimension d and a context window size of w neighbors in each direction, we train an MLP that takes as input a $(d \cdot (2w + 1) + 1)$-dimensional vector and outputs the probability that the target paragraph in this context window is the first paragraph of a heading at the given depth.

Note that this information, analogously to the segmentation vectors, suffices to rebuild the heading tree in full. Accordingly, we infer recursively using this MLP to build the heading tree from the top down by allowing the model to predict the section divisions at each depth. To execute this procedure efficiently, we divide it into three steps:

1. The dataset is first transformed into a set of $(context\,window, depth, boolean)$ triplets. For each paragraph in an article, we recursively search for which headings that paragraph begins. For the depths that a target paragraph starts a heading at, we form a context window with the target in the center and assign a positive label. All other context windows and depths are assigned negative labels. During training, we compare the model predictions to labels with a binary cross-entropy loss.

2. We generate a root node at depth 0 for the tree. For each target paragraph in the sequence, we use the MLP to predict whether that paragraph starts a heading at depth 1 via a threshold probability. For each subsequence of paragraphs between breaks, we create a heading node in the tree with depth 1 and recur.

3. In general, for any given subsequence of the article and parent node, we use the MLP to determine where we need to insert new subheading nodes; for each new insertion, we recur. Otherwise, we create leaf nodes to represent the paragraphs, and we are done.

We tried two different MLP architectures: a small one with layer dimensions of $[(d \cdot (2w + 1) + 1), 1024, 256, 64, 1]$ and a window size of $w = 2$, and a large one with layer dimensions of $[(d \cdot (2w + 1) + 1), 5096, 1024, 256, 64, 1]$ and a window size of $w = 4$. The smaller models had a paragraph embedding dimension of $d = 256$ and the larger models had dimen-

sions of $d = 512$, with the exception of the Doc2Vec-based models which always use $d = 256$.

For the FastText and SimCSE-based MLP models, we required a bidirectional LSTM projection layer to project the sequences of word or sentence embeddings to paragraph embeddings. These projections were trained jointly with the feed-forward parts of the model, and were used for the greedy decoder and end-to-end model, which we discuss below.

End-to-end transformer. Given fixed-length paragraph embeddings, we apply a transformer decoder and then a linear projection to a vector output, either as outline or segmentation vectors. The heading tree is then reconstructed algorithmically from the vector output.

Our transformer decoder had 3 layers, with a hidden dimension of 128 and a feed-forward dimension of 1024. We used 4 attention heads. We trained this model using Doc2Vec embeddings and projected SimCSE embeddings.

3.5 Human annotations

To ground our model results and formulate a human benchmark for the task, we measured human performance by building an interactive web application to create an interface for participants to solve the article outlining task. This app can also serve as a hub to generate interest in and crowdsource data for this task, as well as possibly publicize up-to-date model performance. The app is live at `https://outline-turk.herokuapp.com/`.

App overview. The app uses a concise nested-bracket representation of the heading tree for user submissions. It continually generates a preview of the currently proposed outline for the user to see, which updates whenever the outline or article number fields are edited. The outline field is also pre-filled for each new article with an initial degenerate outline in the form $[1, 2, ..., n]$, which makes it easier for the user to make their desired changes by simply inserting brackets as well as providing the total number of paragraphs for the given article.

The app also performs real-time evaluation of user submissions, and provides a live results page that computes and shows each user's mean JPL score over their submissions. Each user JPL score is paired with a model JPL score using cached results from one of our better-performing models, Greedy-Doc2Vec-128, averaged over the same articles as the user's submissions. This is to provide an estimate of model performance over the articles the user tackled, normalized to the difficulty of those articles, since article difficulty is relatively high variance especially if a user only completes a few articles. The competitive aspect of benchmarking each user against the model's respective performance, as well as the format of the results table as a sorted leaderboard, serve to incentivize user participation by gamifying the outlining task (it is also to this end that the app was given a secondary title "Outlinedle," playing off the popular Wordle line of Internet minigames).

App technical details. The front-end is built on Bootstrap with JQuery for interactive elements, and the back-end runs on Flask with Jinja2 templating and reads and writes data to/from a Firebase database. We use AJAX to generate the dynamic preview, and deployed the app for user submissions via Heroku, which provides in-built concurrency.

4 Results

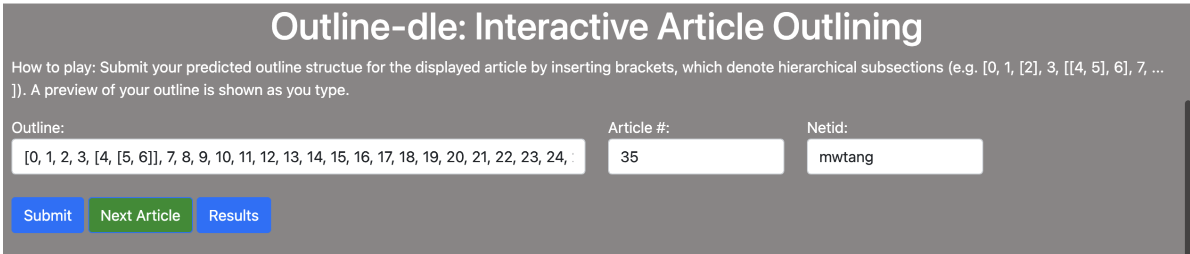
We begin by discussing overall performance on the testing splits of the Wikitext data in sections 4.1-4.3. After, we delve deeper into possible differences in model performance on different types of articles in 5.1, as well as a deeper analysis of the embedding spaces generated by different techniques in 5.2. We finish with an ablation of our chosen evaluation metric in 5.3.

4.1 Greedy decoder

We observe that the greedy decoder, which does not contain any trainable parameters (aside from the LSTM projection layers) performs quite well. The bottom-up dynamic programming nature of the algorithm is suited quite well to the hierarchical structure of the output. The way in which the greedy algorithm groups nearby paragraphs into headings by similarity at each level feels quite natural and comparable to how a human being might approach the problem.

Interestingly, the encoding type that performs best with the greedy decoder uses averaged SimCSE sentence embeddings, as opposed to the projected SimCSE embeddings like one might expect. We discuss this and our general results on the nature of the embedding spaces further in 5.2.

One important detail to note is that the greedy decoder is quite fast: the majority of inference time on an



(3) For the 2003 – 04 season , the Rockets moved into their new arena , the Toyota Center , with a seating capacity of 18 @,@ 500 . During the 2007 – 08 NBA season where the team achieved a 22 @-@ game winning streak , the Rockets got their best numbers to date , averaging 17 @,@ 379 spectators . These were exceeded once James Harden joined the team in 2013 . The Rockets averaged 18 @,@ 123 spectators during the 2013 – 14 season , selling out 39 out of the 41 home games . 2014 – 15 had even better numbers , with 40 sellouts and an average of 18 @,@ 230 tickets sold .

[HEADING]

(4) When the Rockets debuted in San Diego , their colors were green and gold . Road uniforms featured the city name , while the home uniforms feature the team name , both in a serifed block lettering . This was the only uniform design the Rockets would use throughout their years in San Diego . The Rockets ' first logo featured a rocket streaking with a basketball surrounded by the team name .

[HEADING]

(5) Upon moving to Houston in 1971 , the Rockets replaced green with red . They kept the same design from their San Diego days , save for the change of color and city name . The logo used is of a player with a spinning basketball launching upward , with boosters on his back , leaving a trail of red and gold flames and the words " Houston Rockets " below it .

(6) For the 1972 – 73 season , the Rockets introduced the famous " mustard and ketchup " logo , so dubbed by fans , featuring a gold basketball surrounded by two red trails , with " Houston " atop the first red trail and " Rockets " (all capitalized save for the lowercase ' E ' and ' T ') in black surrounding the basketball . The initial home uniforms , used until the 1975 – 76 season , features the city name , numbers and serifed player name in red with gold trim , while the away uniforms feature the city name (all capitalized except for the lower case ' T ' and ' N ') , numbers and serifed player name in gold with white trim .

(7) In the 1976 – 77 season , the Rockets modified their uniforms , featuring a monotone look on the Cooper Black fonts and white lettering on the road uniforms . On the home shorts , the team logo is located on the right leg , while the away shorts feature the team name wordmark on the same location . With minor modifications in

Figure 2: Outline-dle web app main interface `https://outline-turk.herokuapp.com/`

Outline-dle Leaderboard

Overall User APPL Score (all users): **2.3475**
Overall Model APPL Score (articles from all user submissions): **1.6304**

Guide:

- **APPL (All-Pairs Paragraph LCA) Score:** mean squared least common ancestor distances in heading trees averaged over all pairs, normalized by the number of paragraphs. Lower is better.
- **User Score:** average APPL score from user submissions
- **Model Score:** average APPL score from model predictions over the same articles as the user submissions, using Doc2Vec embeddings + recursive greedy decoder model.
- **Overall User Loss:** mean of User Loss over all users
- **Overall Model Loss:** mean of Model Loss over all users

Netid	User APPL Score ↓	Model APPL Score
mwtang	**0.2835**	1.2857
jlding	**0.4551**	1.5063
andrewmi	**0.5714**	1.2381
jiatongy	**0.8009**	1.3333
naive	3.7259	**1.7551**

Figure 3: Outline-dle web app results interface `https://outline-turk.herokuapp.com/results`

Model	JPL Score
Greedy-Doc2Vec	1.316
Greedy-Projected FastText	1.431
Greedy-Averaged SimCSE	**1.273**
Greedy-Projected SimCSE	1.319
MLP-Doc2Vec	1.256
MLP-FastText	1.302
MLP-SimCSE	**1.221**
End-to-End-Doc2Vec	4.732
End-to-End-SimCSE	4.732
Human	**0.469**

Figure 4: Results from all model architectures

article is spent on embedding the paragraphs, which the greedy decoder only does once.

4.2 Recursive MLP

The recursive MLP decoder performs the best. We expect that this is because the recursive decoding process lends itself nicely to the hierarchical structure of the output. At the same time, however, the problem of classifying paragraphs by whether they start a heading at a particular depth is a very tractable problem for a lightweight MLP to handle, especially when it is given the context of nearby paragraphs. We did not observe any major difference in the recursive MLP decoder's performance between the two sizes with any of the 3 encoder setups; so the results shown in the table are those gathered with the smaller models.

We observe that the encoding type that performs best with the recursive MLP decoder uses the projected SimCSE sentence embeddings. This makes sense, because SimCSE has been shown to produce state-of-the-art, semantically meaningful sentence embeddings. Via the LSTM projection layer, we allow the MLP to make more flexible use of the information within each paragraph than the rigid nature of Doc2Vec, for example. We discuss our results on the nature of our embedding spaces further in 5.2.

An important element that we observed is that the recursive MLP decoder takes much longer than the greedy decoder to evaluate: specifically, it takes about 1 second per article to generate a tree. This is because at each depth in the recursion, the MLP model must be again inferenced at the same context windows, but with a different depth as input.

4.3 End-to-end transformer

The end-to-end models perform the worst. They produce degenerate output, in which they predict the same segmentation sequence for any input. After investigating, we found that the degenerate output correlates strongly with the mean of the segmentation vectors over the training set. This behavior persists throughout various changes in architecture, hyperparameters, overfitting conditions, etc. We attempted to correct for this via normalization by mean and standard deviation, but it continues to predict a constant output that is correspondingly normalized. In the end, we observed that the other methods of decoding a sequence of paragraph embeddings to form a heading tree are more naturally suited to the hierarchical structure of the output. The sequence-to-sequence nature of the transformer required us to represent trees as sequential output; while the outline and segmentation vectors appear reasonable representations at the surface, it seems that it is difficult for a transformer to make meaningful progress. Perhaps more research can be done into how to encode a tree into a sequential representation that is more learnable.

5 Discussion

5.1 Performance over Different Article Lengths and Depths

We hypothesized that the number of paragraphs and the level of nested headings would impact model performances, as observed in human annotation scores. Below are our visualizations of SimCSE JPL scores over the testing set distributions.

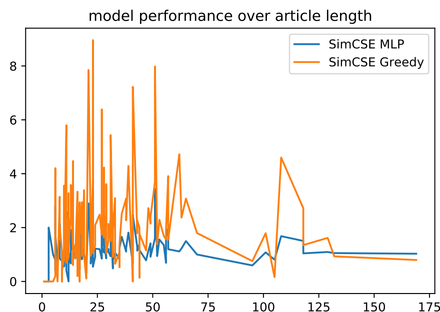


Figure 5: SimCSE models performance over number of paragraphs in article

As shown in Figure 5, despite their similar overall performance, the MLP decoder is more robust to long article lengths. The long tail is due to testing set distribution of few but extremely long articles. Performances before lengths of around 75 paragraphs are therefore more representative.

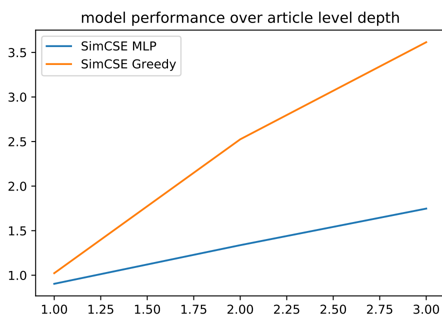


Figure 6: SimCSE models performance over article depth

Similarly, Figure 6 shows that the MLP decoder is also more robust against article depth (height of the heading tree for a given article), but both performed worse with higher article depths. Since article levels are discrete, we computed the mean JPL loss score over different article depths for each model, and plotted the points in figure 6. The MLP and the greedy decoder achieved similar scores at shallower documents, but the MLP significantly outperforms the greedy decoder when the article has deeper nested headings.

5.2 Embedding

Each of the embedding methods we used (FastText, SimCSE, and Doc2Vec) are dominant at their respective scales (words, sentences, and paragraphs). We adapted these embedding methods to our use case in order to embed paragraphs from articles, and in so doing have generated latent spaces that are different than what the embedding methods were initially designed for. In particular, note that what we desire from paragraph embeddings has some structural and relational information within it: the greedy decoder, for example, uses the cosine similarity between paragraph vectors to determine if they belong under the same subheading. There is more information that could be useful to this subtask than simply semantic meaning, which these embedding methods are developed for. In addition, we change the scales at which FastText and SimCSE operate via a projection. For all these reasons, we wish to evaluate how our encoding techniques behave in our target domain of paragraphs within articles.

To accomplish this, we can visualize the generated paragraph encodings via dimensionality reduction. We take paragraphs from several ($N = 12$) articles and label them according to which article they came from. We then apply principal component analysis (PCA) to reduce the vectors to have 50 dimensions, on top of which we apply t-distributed stochastic neighbor embedding (tSNE) to reduce the vectors to 2 plottable dimensions. We look for clusters around particular articles and separation between distinct articles. The results for 4 different encoding techniques are graphed below.

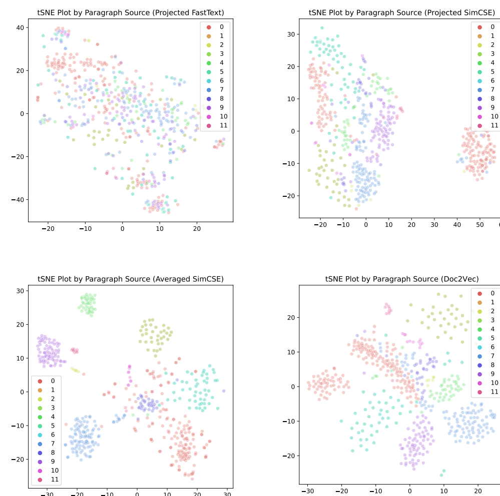


Figure 7: Paragraph embeddings with tSNE reduction visualized for: LSTM-Projected FastText, Averaged SimCSE, LSTM-Projected SimCSE, and Doc2Vec.

We observe from the tSNE graphs in Figure 7 some interesting phenomena. Importantly, we see that the averaged SimCSE (bottom left) embeddings form very nice clusters consisting of paragraphs from the same article while maintaining a good distance between articles. The Doc2Vec (bottom right) embedding space is similarly nice, but with less separation of clusters. The projected FastText (top left) embedding space is very spread, and the projected SimCSE has decent clustering, but with a distinct lack of uniformity. In general, we see that the averaged SimCSE embedding space is the best in terms of both uniformity, separation of clusters, and alignment. We discuss some potiential conclusions to be drawn from this evaluation in 6.

5.3 Evaluation metrics

To compare different evaluation metrics, we use a heuristic that if we slightly perturb a heading tree by randomly adding and deleting headings, and compose this perturbation sequentially, we should find that for a given loss function, the loss between a tree and the original tree should have high Spearman (rank) correlation with the number of times the tree has been perturbed from the original.

Formally, we perturbed by uniformly randomly deleting each existing heading with some small probability p_{remove} and then uniformly randomly adding a heading on every possible range of paragraphs that are currently under the same heading with some small probability p_{add}, where we compute these probabilities such that the expected number of removed and added headings in each round of perturbation is exactly 1.

Since perturbation is noisy, we generated 10-step perturbation sequences over 10 random starting heading trees from our training data, each sequence run 50 times, and computed the mean Spearman correlations for each evaluation metric:

Metric	Spearman Correlation
APPL	0.771 ± 0.239
JPL	$\mathbf{0.843 \pm 0.204}$
Elastic$_{\alpha=0.5}$	0.807 ± 0.235
Edit	0.842 ± 0.203

We found that JPL performed the best overall, closely followed by Edit. Thus, we chose to use JPL as our evaluation metric. All correlations had a p-value under 0.05.

5.4 Qualitative analysis

To sanity-check and visualize our model predictions, we print some of the predicted heading trees produced by one of our better-performing models, the Greedy-Doc2Vec alongside their ground truth counterparts. One example is shown in 8, with snippets of each paragraph printed for illustration. Qualitatively examining these trees, we see that the model successfully picks up on some of the paragraph divisions from the ground truth tree – in the figure, it correctly groups together paragraphs beginning with "In September 2006..." and "In February 2011," as well as the first three paragraphs beginning with "Dan Dugan...," "In his youth...," "Dugan first recorded...," among other correct headings.

At the same time, since the model predicts on local groups of articles, it misses some common patterns among the article heading trees. For example, it is customary for Wikipedia articles to begin with 1-3 paragraphs which are always at the highest depth level; in the case of 8, this is the first three.

5.5 Human performance

We collected 30 human predictions on different articles from different participants and found that humans perform significantly better than even our best model, with a JPL score of 0.469 compared to 1.273 from Greedy-Averaged SimCSE 4.

Although we initially thought the task might be very difficult for humans to solve, this is evidently not the case. However, it is worth noting that these are somewhat biased results: humans often choose to skip long articles, which are often the examples the models perform the worst on, and the dataset of Wikipedia articles is also one that the humans already have extensive exposure to, so they often employ tricks like recognizing familiar-looking articles such as ones on TV show episodes, which have a very common Introduction-Plot-Reception format. The website design and input format also worked well, as human participants had little trouble picking up the relatively unusual task.

6 Conclusion

We approach a novel problem, article outlining, with an encoder-decoder framework made of composable parts, and experiment with variations on several different encoder embeddings: Doc2Vec, SimCSE, and FastText. We experiment with and ablate different techniques for embedding and decoding: the best-performing model overall is the MLP with SimCSE paragraph embeddings generated as an average of frozen sentence embeddings 4. Generally, SimCSE is our strongest embedding, as using it with averaging as well as

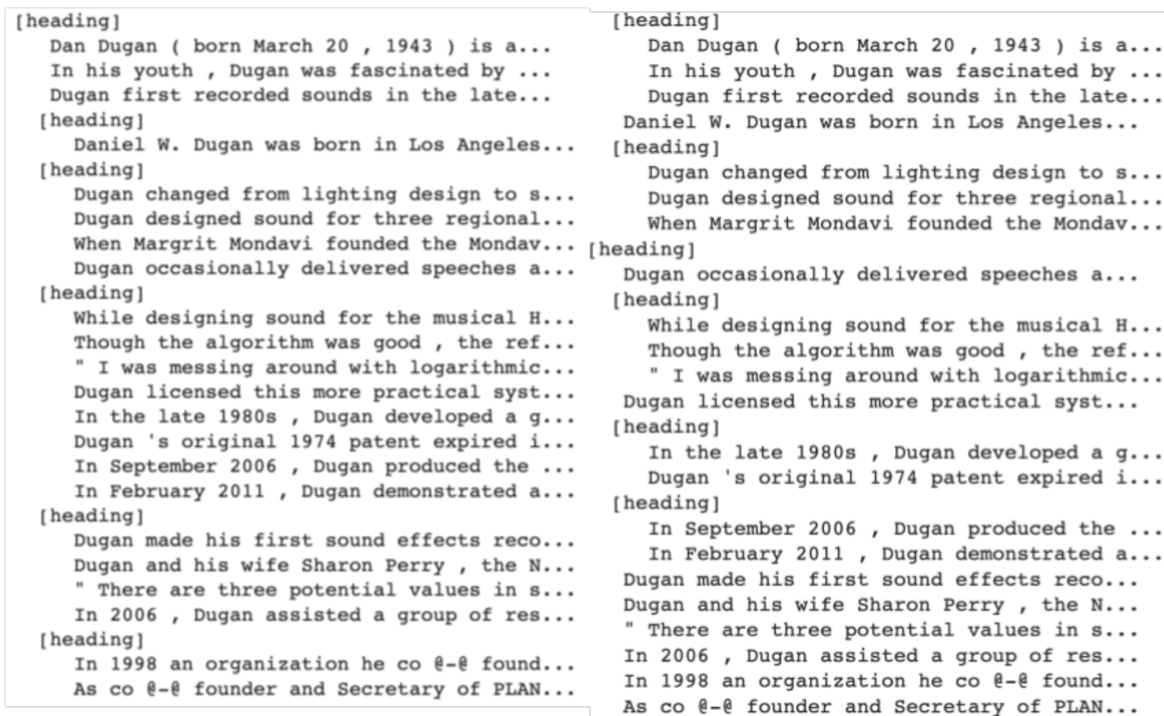


Figure 8: Ground truth (left) and predicted (right) heading trees for example article “Dan Dugan (audio engineer)”

with a projection via LSTM lead to some of the best results among all of the different model-embedding combinations, and the robustness of this embedding is supported by our analysis of its intrinsic properties 7. We also propose the JPL evaluation metric for comparing trees and substantiate its superiority over other techniques by looking at rank-correlations with sequentially perturbed trees 5.3. Overall, we lay the groundwork for further research on paragraph embeddings and this novel task of article outlining.

7 Future Work

As mentioned previously, our models are limited by both time and computational resources and we were only able to run on a fraction of the training set for the end-to-end model, which still took upwards of 10 hours to train. Future work should train on the entire WikiText dataset, and we expect the outcome embedding would outperform our current best models.

Since few deep learning works involve general tree-structure data, we designed our own labeling and objective functions for this task, which can serve as baselines for more in-depth study of tree-structure data. While our numeric representation of paragraph-relations are straightforward for algorithmic decoders, these label vectors may not suit the demand of deep learning decoders such as transformers.

Finally, the style and structure of Wikipedia articles may differ from documents from other sources. Therefore, future work should also run the proposed models on different document sources to see which generalize better.

8 Acknowledgements

We would like to thank Professor Karthik Narasimhan for his kind feedback and support.

References

[1] Piotr Bojanowski et al. “Enriching Word Vectors with Subword Information”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146. ISSN: 2307-387X.

[2] Tianyu Gao, Xingcheng Yao, and Danqi Chen. “Simcse: Simple contrastive learning of sentence embeddings”. In: *arXiv preprint arXiv:2104.08821* (2021).

[3] Stephen Merity et al. *Pointer Sentinel Mixture Models*. 2016. arXiv: 1609.07843 [cs.CL].

[4] Le Quoc and Mikolov Tomas. “Distributed Representations of Sentences and Documents”. In: *arXiv preprint arXiv:1405.4053* (2014).